UNIVERSIDAD AUTÓNOMA DEL
ESTADO DE QUINTANA ROO

DIVISIÓN DE CIENCIAS, INGENIERÍA Y TECNOLOGÍA

ARQUITECTURA LIMPIA CON MVVM (*MODEL-VIEW-VIEWMODEL*) PARA APLICACIONES MÓVILES ANDROID

TRABAJO MONOGRÁFICO

PARA OBTENER EL GRADO DE

INGENIERO EN REDES

PRESENTA

AARON ENRIQUE CASTILLO GOMEZ

ASESORES

MTI MELISSA BLANQUETO ESTRADA
MTI VLADIMIR VENIAMIN CABAÑAS VICTORIA
DR. JAIME SILVERIO ORTEGÓN AGUILAR
DR. JAVIER VÁZQUEZ CASTILLO
M.S.I. LAURA YÉSICA DÁVALOS CASTILLA

CHETUMAL QUINTANA ROO, MÉXICO, OCTUBRE DE 2022

UNIVERSIDAD AUTÓNOMA DEL
ESTADO DE QUINTANA ROO

DIVISIÓN DE CIENCIAS, INGENIERÍA Y TECNOLOGÍA

TRABAJO MONOGRÁFICO TITULADO

"ARQUITECTURA LIMPIA CON MVVM (MODEL-VIEW-VIEWMODEL) PARA APLICACIONES MÓVILES ANDROID"

ELABORADO POR

AARON ENRIQUE CASTILLO GOMEZ

BAJO SUPERVISIÓN DEL COMITÉ DEL PROGRAMA DE LICENCIATURA Y APROBADO COMO REQUISITO PARCIAL PARA OBTENER EL GRADO DE:

INGENIERO EN REDES

COMITÉ SUPERVISOR

SUPERVISORA: _____
MTI MELISSA BLANQUETO ESTRADA

SUPERVISOR: _____
MTI VLADIMIR VENIAMIN CABAÑAS VICTORIA

SUPERVISOR: _____
DR. JAIME SILVERIO ORTEGÓN AGUILAR

SUPERVISOR SUPLENTE: _____
DR. JAVIER VÁZQUEZ CASTILLO

SUPERVISORA SUPLENTE: _____
M.S.I. LAURA YÉSICA DÁVALOS CASTILLO

CHETUMAL QUINTANA ROO, MÉXICO, OCTUBRE DE 2022

UNIVERSIDAD AUTÓNOMA DEL ESTADO DE QUINTANA ROO
ÁREA DE TITULACIÓN

UNIVERSIDAD AUTÓNOMA DEL ESTADO DE QUINTANA ROO
DIVISIÓN DE CIENCIAS, INGENIERÍA Y TECNOLOGÍA

Resumen

A nivel mundial el uso del teléfono móvil se ha propagado entre las personas y es uno de los dispositivos más utilizados para la comunicación, el trabajo y el entretenimiento; para los fines antes mencionados el usuario accede a diversas aplicaciones de software.

Las empresas y desarrolladores de aplicaciones trabajan día con día para poder ofrecer diferentes soluciones informáticas para cubrir la demanda de software de los usuarios, por lo que es importante para los programadores aprovechar las metodologías, técnicas y herramientas tecnológicas que le permiten llevar a cabo el proceso de desarrollo de software.

La arquitectura de software determina la estructura del proyecto, cómo se organiza el contenido de un proyecto de software. Esto es de suma importancia para el desarrollador, pues al trabajar con diferentes proyectos, necesita de mayor facilidad y flexibilidad para ubicar archivos, el código para hacer algún cambio al diseño de la pantalla, o dónde modificar las reglas de negocio, etc.

En esta monografía, se aborda qué es la arquitectura de software, los patrones de diseño para móviles y la arquitectura limpia a través de un proyecto para una aplicación móvil en Android.

Agradecimientos

Agradezco a mis padres por siempre apoyarme en las decisiones que he tomado a lo largo de la carrera, los consejos, experiencias y oportunidades que me han proporcionado. Por los ánimos que siempre me han dado aun pasando por momentos complicados y guiarme para poder cumplir mis objetivos, sueños y hacerme la persona que soy.

A mi familia, tíos, primos y abuela que de igual manera siempre me apoyaron y motivaron durante toda la carrera y estuvieron ahí cuando los necesitaba.

A mis amigos Edgar, Osmar y Abraham que durante toda la carrera estuvieron conmigo clase a clase apoyándome y ayudándome en temas en los que no era del todo bueno o cosas que no terminaba de entender.

De igual manera a la MTI Melissa Blanqueto Estrada por ser mi asesora, estar siempre ayudándome, aconsejándome y apoyándome durante el tiempo que estuve elaborando este trabajo y al MTI Vladimir Veniamin Cabañas Victoria por igual apoyarme durante el proceso de titulación.

Dedicatoria

Dedico este trabajo a mi familia, con especial agradecimiento a mis papás por siempre estar tras de mí, pendientes y motivándome para cumplir mis metas, terminas mis estudios y poder ser la persona en la que me he convertido. Estar siempre para mí, cuando siempre los necesite, y aunque haya habido adversidades, me ayudaron para tener este gran logro.

De igual manera dedico a mi abuela que siempre me da conejos, regaños y me demuestra su apoyo incondicional, y está ahí para guiarme día a día.

Contenido

Figuras

1. Introducción

Actualmente el uso de dispositivos móviles es masivo, la telefonía celular se ha extendido a gran parte de la población y por lo tanto, el desarrollo de aplicaciones móviles tiene gran relevancia para las organizaciones y personas que crean software y ofrecen aplicaciones al usuario para interactuar en los diversos contextos de su vida cotidiana.

Dentro del ecosistema de los sistemas operativos para dispositivos móviles, Android ocupará el 87.4% del mercado global en el 2023 (Statista, 2022); por lo tanto, es de interés para las empresas y programadores independientes el desarrollo de software para los consumidores. Un aspecto para considerar al momento de desarrollar una aplicación móvil es la arquitectura que tendrá el proyecto, ya que esta determinará su estructura, la ubicación de archivos, módulos, secciones, etc. Esto es relevante ya que el desarrollador debe ubicar, por ejemplo, el archivo donde se encuentran las reglas de negocio. En la mayoría de los casos el proceso de desarrollo se lleva a cabo en equipo y cualquier desarrollador que trabaje en el proyecto y no tenga experiencia en éste, podrá encontrar con mayor facilidad lo que necesita.

Existen varias opciones para la arquitectura de software de diferentes tipos de aplicaciones; específicamente en aplicaciones para Android, los patrones de diseño que más predominan son MVP (*modelo-vista-presentador)*, MVC (*modelo-vista-controlador*) o MVVM (*modelo-vista-vista modelo*). Los patrones antes mencionados facilitan el trabajo a los programadores, organizando el proyecto y aumentando la agilidad durante el desarrollo de software.

En este trabajo monográfico se aborda la integración de la arquitectura limpia con el patrón de diseño MVVM. Primero se define lo que es la arquitectura de software para poder entender mejor la arquitectura limpia, también se presentan los patrones de diseño para aplicaciones Android, sus ventajas y desventajas; por último, se

implementan a través de un proyecto de ejemplo la arquitectura limpia y el patrón MVVM.

Objetivo general

Explicar el uso del patrón MVVM en el desarrollo de aplicaciones móviles para Android y su implementación en conjunto con una arquitectura limpia.

Objetivos Específicos

- Definir a qué se refiere la arquitectura limpia y su funcionamiento.
- Definir qué es el patrón MVVM para aplicaciones móviles Android.
- Describir las ventajas y desventajas del patrón de desarrollo de software MVVM en comparación a otros patrones.
- Explicar cómo implementar MVVM con una arquitectura limpia.

2. Marco teórico

Arquitectura de software

Arquitectura es un término que se refiere a diversos aspectos que tiene relación con las TI. Software Engineering Institute (SEI) dice que la arquitectura de software se puede definir como “las estructuras de un sistema, compuestas de elementos con propiedades visibles de forma externa y las relaciones que existen entre ellos.” (Bass, Clements, & Kazman, 2013)

Los patrones en la arquitectura de software han existido en la comunidad de software desde al menos principios de la década de 1990. Los desarrolladores de software se dieron cuenta del papel crucial que juega la arquitectura de software en el desarrollo, mantenimiento y evolución exitosos de su sistema de software. Un buen diseño de arquitectura de software puede conducir a un producto que satisfaga las necesidades del cliente y que pueda actualizarse fácilmente, mientras que una arquitectura inadecuada puede tener consecuencias desastrosas que pueden llevar al retiro de un proyecto (Hanmer, 2013).

Los autores de patrones de software han estado escribiendo patrones que documentan sus soluciones comprobadas para que en un futuro otros desarrolladores se beneficien de su experiencia. El objetivo de los patrones y arquitectura es acelerar el desarrollo; permitir al desarrollador avanzar, sabiendo que una arquitectura en particular lo ayudará en lugar de obstaculizarlo; y, en última instancia, darle el tiempo que necesita para resolver problemas nuevos.

El desarrollo de software a menudo solo se enfoca en la funcionalidad sin importar cómo el software se vuelve mantenible para que otros desarrolladores puedan desarrollarlo fácilmente.

Es de suma importancia la arquitectura en los proyectos de desarrollo, dado que delimita como se estructura, tiene gran impacto por la capacidad para satisfacer y

mejorar atributos de calidad del sistema. Uno de estos atributos es el *desempeño* que el sistema debe tener, que hace referencia al tiempo de respuesta durante el uso de este. Otro atributo es la *usabilidad*, que es que tan fácil es usar el sistema para un usuario, realizar tareas en él. La *modificabilidad* igual es un atributo importante, que igual es de suma importancia para el desarrollador, determina qué tan sencillo es poder agregar cambios y nuevas funcionalidades en el sistema.

Por otro lado, la arquitectura de software aparte de tener como beneficio los atributos de calidad, tiene gran importancia al orientar el desarrollo. Una de tantas estructuras que la conforman se encarga de dividir el sistema en diversos componentes que tendrán que ser programados por personas o equipos de desarrollo. Es esencial el identificar la forma de asignación de trabajo, esto para apoyar la planeación de tareas del proyecto, así como su desarrollo. (Cervantes, 2010)

Los diseños de arquitectura que se elaboran en una empresa se pueden reutilizar para desarrollar otros sistemas distintos. Esto trae como beneficio la reducción de costos y tener en aumento la calidad de los programas, aún más si estos vienen de sistemas que resultaron en éxito (Juarez, 2020).

Arquitectura de software es un término que se ha vuelto común en los últimos años para describir la relación entre varios aspectos de los sistemas de software, aunque no existe una definición universalmente aceptada. La comunidad de modelado de confiabilidad también usa el mismo término para describir la interacción entre módulos para predecir la confiabilidad. Algunas definiciones o descripciones por diversos autores son las siguientes:

- Una manera de poder definir el sistema en términos de componentes computacionales y sus diversas interacciones (Shaw & Garlan, 1996).
- La estructura de todos los componentes de un sistema/programa, sus interrelaciones, los principios y directrices que llevan a su evolución en el tiempo (Garlan & Perry, 1995).

- El diseño de un sistema, que se integra en cuestiones separadas oeri interesantes de un sistema, como disposiciones independientes para una evolución independiente y apertura combinadas con confiabilidad y rendimientos generales (Szyperski, 2002).
- La arquitectura de software de un sistema es el conjunto de estructuras necesarias para razonar sobre el sistema, las cuales comprenden elementos de software, relaciones entre ellos y propiedades de ambos. (Bass, Clements, & Kazman, 2013).
- La especificación completa y detallada de la interfaz de usuario, “donde la arquitectura dice lo que sucede, la implementación dice cómo se hace que suceda” (Brooks, 1995).

En términos generales, se puede concluir básicamente el término *arquitectura de software* no tiene un significado establecido, pero se puede explicar o entender como la parte que define la estructura de un proyecto, programa o sistema, desde sus diferentes componentes, capas y módulos, todo esto para que el desarrollador pueda tener un fácil acceso a estos, le sea sencillo poder entender la estructura en su totalidad y cada uno de los componentes del sistema en el que trabajara, y facilitarle el poder resolver problemas e implementar cosas nuevas a este.

3. Arquitectura limpia

En los últimos tiempos se ha tenido una gran cantidad de ideas con respecto a la arquitectura se software en los sistemas, como son: Arquitectura hexagonal, arquitectura cebolla, DCI, BCE, entre algunos otros.

Estas arquitecturas son similares, aunque varían un poco en sus detalles. Todos tienen un mismo objetivo, que es la división de objetivos de cada sección. Todos logran esto dividiendo en capas todo el software desarrollado. Cada uno tiene por menos una capa para casos de uso o reglas de negocio y otra para interfaces. (Vespa, 2019)

En el caso de la arquitectura limpia, ofrecerá lo siguiente (Martin, 2018):

- **Independiente de *frameworks*:** no depende de alguna plugin o biblioteca externa de software cargada de funciones, esto permite utilizar dichos complementos como herramientas en vez de tener un sistema con sus limitaciones.
- **Comprobable:** las reglas de negocio se pueden probar sin necesidad de tener o utilizar la interfaz de usuario, la base de datos, el servidor o cualquier otro elemento externo ya que este es independiente.
- **Independientes de la interfaz de usuario:** la interfaz de usuario puede sufrir cambios sin problemas alguno, esto sin tener que realizar modificaciones al resto del sistema, una interfaz de usuario web se podría cambiar con una de consola, sin necesidad de cambiar las reglas de negocio.
- **Independiente de las bases de datos:** se puede hacer un cambio del tipo de bases de datos utilizadas sin alterar el sistema o alguna otra capa. Podría cambiar de una base de datos Oracle o SQL Serve por MongoDB.

Regla de dependencia

Cada uno de los círculos de la Figura 1 representa los diferentes ámbitos en el desarrollo de software. Básicamente, cuanto más avance, mayor será el nivel del software. Los círculos exteriores se tratan de mecanismos. Los círculos internos son políticas y reglas.

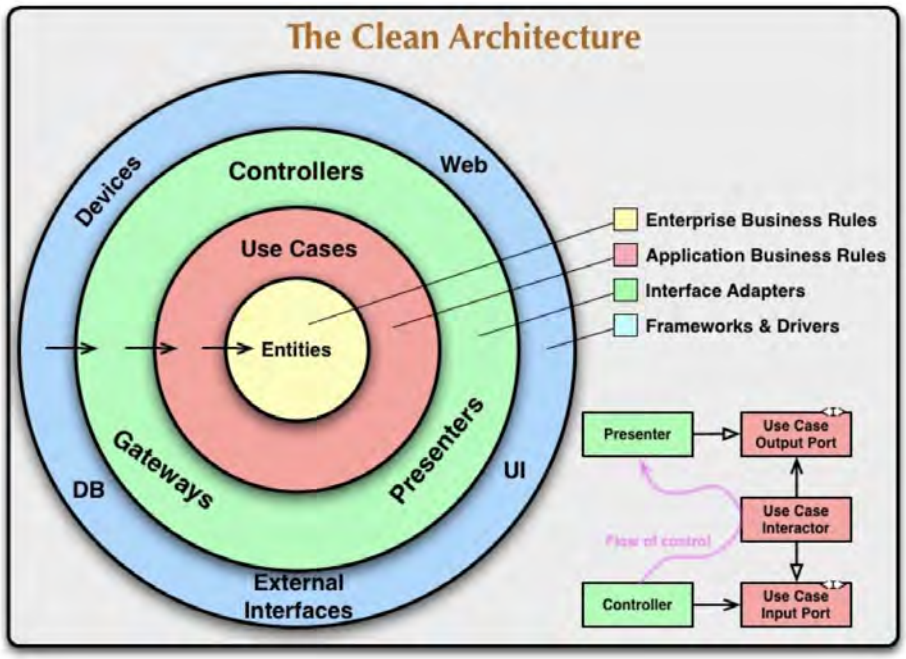


Figura 1 Clean Architecture (Martin, 2018)

La *regla de dependencia* es la más importante y principal que hará que esta arquitectura funcione, esta regla principalmente se refiere a que las dependencias de código solo pueden apuntar hacia adentro. (Samuel, 2019)

Usando la regla en la Figura 1, sería que nada en una capa o círculo interior puede saber absolutamente nada sobre un círculo exterior. Hablando específicamente de código, cualquier cosa declara en una capa exterior no debe ser utilizado por el código en una capa interior. Eso incluye, clases, variables, funcione o cualquier entidad de software creada.

De igual forma, los formatos de los datos situados en un círculo exterior no deberían ser utilizados en uno interior, mayormente si estos formatos son generados por un marco en un círculo exterior.

Básicamente, no se quiere que algo del algún circulo exterior impacte a un círculo interior.

Entidades

En esta capa se encuentran las reglas comerciales/negocio de la empresa. Una entidad puede ser un objeto con diferentes métodos o puede ser un conjunto de funciones y las estructuras de datos. Todo esto para que las entidades puedan ser utilizadas por muchas aplicaciones diferentes en la empresa (Martin, 2018).

En caso de no ser un sistema de una empresa y es una sola aplicación, las entidades son los objetos comerciales o reglas de negocio de la aplicación. Contiene las reglas más generales y de alto nivel.

Son los que menos posibilidades tienen de cambios cuando algo externo cambia, por ejemplo, no se esperaría que las reglas cambien si se afecta la navegación del sistema o la seguridad, puesto que esto no cambia las reglas establecidas en los objetos. Ningún cambio operativo en ninguna aplicación en particular debería afectar la capa de entidad (Martin, 2018).

Casos de uso

Todo el código de esta capa contiene reglas de negocio ***específicas de la aplicación.*** En general son todos los casos de uso que se implementan en la aplicación. Estos casos de uso determinan el flujo de datos desde y hacia las entidades, dirigen a esas entidades para que se pueda hacer uso de sus reglas de

negocio decretadas por la empresa para lograr los objetivos determinados en los casos de uso (Martin, 2018).

Se espera que cambios en esta capa no afecten a las entidades, de igual manera no se espera que cambios en capas exteriores afecten a esta, como bases de datos, interfaz de usuarios, etc. Esta capa está aislada de dichas preocupaciones.

Por otro lado, si existen cambios en el funcionamiento de la aplicación, se tiene previsto que afecte los casos de uso, por lo tanto, todo el software de esta capa. Si algún detalle de los casos de usa cambia, parte del código de esta capa tendrá afectaciones y cambios.

Adaptadores de interfaz

El software desarrollado en esta capa los formatos de datos más convenientes utilizados en los casos de uso y entidades, a un formato más adecuado para algo externo como las bases de datos o la web.

Como se menciona, ningún circulo al interior de uno debe saber “qué es lo que pasa” en él, en este caso, ningún círculo o capa dentro de este, (casos de uso y entidades), deben saber sobre la base de datos. Si la base de datos es SQL, todo lo que tenga que ver con SQL debe estar presente únicamente en esta capa y, en particular, a las partes de esta capa que tienen que ver con la base de datos (Martin, 2018).

En esta capa también existen adaptadores necesarios para la conversión de datos en algún formato externo, a formatos utilizados internamente para las entidades y casos de uso.

Marcos y controladores

La última capa se compone generalmente de marcos y herramientas como la base de datos, el marco web, *frameworks*, complementos, *plugins*, etc. Normalmente, no se suele escribir mucho código en esta capa, aparte del que se comunica con la siguiente capa hacia adentro, también conocido como código pegamento, que lo que hace es “adaptar” diferentes partes del mismo para que sean compatibles. (Martin, 2018)

En esta capa se sitúan todos los detalles, tales como la web y la base de datos. Estos detalles se mantienen en el exterior donde no pueden ocasionar daños.

Básicamente, lo que busca la arquitectura limpia es poder independizar por capas o módulos todos los componentes de un sistema de software, que sean independientes entre sí, para poder tener un gran poder en el mantenimiento y

estabilidad, que resulte sencillo realizar cambios y más que nada que si es necesario realizar o modificar algo en una capa, esta no afecta las demás.

4. Patrones de desarrollo en aplicaciones móviles Android

En los últimos años el desarrollo de software no ha sufrido demasiados cambios, los desafíos siguen siendo básicamente los mismos: tener una arquitectura flexible, escribir código limpio, que el este pueda probarse y validarse, tener una correcta y ágil experiencia de usuario, etc.

La mayoría los problemas tenían solución cuando Android ingresó al mercado. Los desarrolladores de aplicaciones de escritorio y web contaban con una base de conocimientos extensa. Esas soluciones llegaron a funcionar para Android, pero no resultaban ser las mejores.

Android se basó en Java en un principio y normalmente el patrón MVC era mayormente el implementado en las aplicaciones, hasta llegar a la actualidad donde es más utilizado *Kotlin* en conjunto al patrón MVVM.

Como se mencionaba anteriormente, un patrón de arquitectura da modularidad a los archivos del proyecto y asegura que todos los códigos se cubran en las pruebas unitarias. Facilita a los desarrolladores la tarea de mantener el software y ampliar las características de la aplicación en el futuro.

En el mundo del desarrollo de aplicaciones móviles Android existe una gran variedad de patrones o arquitecturas de software para los proyectos de aplicaciones. Entre los más utilizados se encuentran el *Model View Controller* (MVC), *Model View Presenter* (MVP), *Model View View Model* (MVVM).

MVVM y MVP son los que actualmente más se utilizan dejando de lado MVC, ya que estos a diferencia de MVC son más sencillos de testear, así como su

modularidad y flexibilidad es mayor ya que MVC tiene un acoplamiento mayor entre los módulos, lo que reduce su grado de flexibilidad y mantenibilidad.

Como se menciona MVVM y MVP son patrones de desarrollo más utilizados, a continuación, hablaremos, definiremos y entenderemos su funcionamiento, diferencias, ventajas, desventajas y porque utilizar uno sobre otros.

MVP

MVP es procedente del conocido MVC (*Model View Controller*) y uno de los patrones más utilizados para organizar la capa de presentación en las aplicaciones Android. Permite separar la capa de presentación de la lógica, para que todo el funcionamiento de la interfaz de usuario sea completamente independiente de la representación en pantalla.

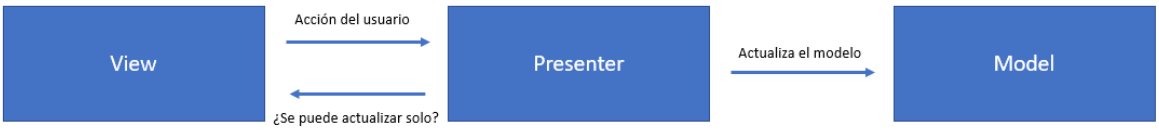


Figura 2. Patrón MVP

Model

Model-View-Presenter (MVP) es un patrón de arquitectura para la capa de presentación de las aplicaciones de software. El patrón se desarrolló originalmente en Taligent en la década de 1990 y se implementó por primera vez en C++ y Java.

"El *Model* contiene la información con la que el sistema trabaja, proporcionándosela a la vista para que pueda mostrarla y permitiendo realizar cambios en ella desde el controlador." No tiene conocimiento sobre la interfaz.

El *Model* es responsable de manejar la lógica del dominio (reglas de negocio del mundo real) y la comunicación con la base de datos y las capas de red. No está vinculado a la vista ni al controlador, y gracias a esto, es reutilizable en muchos contextos.

También es el componente que preserva los datos, el estado y la lógica de negocios; simplemente expone un grupo de interfaces de servicio a *Presenter* y oculta los detalles internos.

View

La View es la interfaz de usuario, recibe la acción del usuario y el contrato con el *Presenter* para lograr la necesidad del usuario, y luego la View responde al usuario según la información del resultado.

Generalmente es implementada con una *Activity* en Android, esto significa que no solo implementa código de UI, sino también de lógica, esto es lo hace diferente al MVC, ya que este solo contiene exclusivamente interfaz de usuario.

Esta tiene una referencia al *Presenter*, ya que lo único que hará la View es utilizar una función del *Presenter* cada vez que haya una acción del usuario (por ejemplo, hacer clic en un botón). (Leiva, 2020)

Presenter

El *Presenter* se encuentra entre la View y el *Model*, recibe información de la View y pasa los comandos al *Model*. Luego obtiene el resultado y actualiza la View a través de la interfaz contratada.

Es similar al *Controller* de MVC, sólo que no está vinculado a una implementación específica de la View, solo a una interfaz. Esto soluciona los problemas de testabilidad, así como de modularidad/flexibilidad que teníamos con MVC.

El *Presenter* es responsable de actuar como intermediario entre la View y el *Model*. Recupera datos del *Model* y los devuelve formateados a la View. (Leiva, 2020)

Además, a diferencia del MVC típico, decide qué sucede cuando se interactúa con la View. Por lo tanto, tendrá un método para cada posible acción que el usuario pueda hacer. (Leiva, 2020)

MVVM

Fue creado por los arquitectos Ted Peters y Ken Cooper de Microsoft con la premisa de hacer simple el desarrollo de la interfaz de usuario basada en eventos. De igual forma MVVM es conocido como Model-View-Binder, esto específicamente en implementaciones que no contempla la plataforma .NET.

MVVM es una evolución de MVP. Tiene como objetivo lograr un mayor desacoplamiento entre la capa intermedia representada por *ViewModel y la interfaz de usuario*. La comunicación entre éstos dos se realiza exclusivamente mediante notificaciones de enlace de datos bidireccionales.

"El ***Model-View-ViewModel*** es una arquitectura que se ha hecho muy conocida a partir de que Google la convirtiera en su arquitectura oficial al lanzar la guía de arquitecturas." (Kotlin, 2021)

El patrón de diseño MVVM puede separar el negocio y la lógica, aumentar la reutilización del código. Al mismo tiempo, hace que el sistema sea sencillo de desarrollar, probar y mantener. El diseñador y el desarrollador pueden coopera entre sí. Puede mejorar la eficiencia del desarrollo del sistema.

Data *Binding* en Android, proporciona beneficios como facilitar las pruebas y la modularidad del proyecto, disminuyendo la cantidad de código que se tiene que desarrollar para poder conectar View + *Model*.

La principal característica de este patrón es la facilidad con la que se puede separar la lógica de la interfaz de usuario de la lógica de reglas de negocio y el modelo de datos de la aplicación. El ViewModel tiene como principal función convertir los datos del Model aun formato que sea fácil de presentar y manejar. Teniendo esto en cuenta, ViewModel tiene la lógica de presentación de la View.

Básicamente es muy parecido a MVP, lo que hace la diferencia es el uso del *ViewModel*, que es la conexión entre el *Model* y las View.

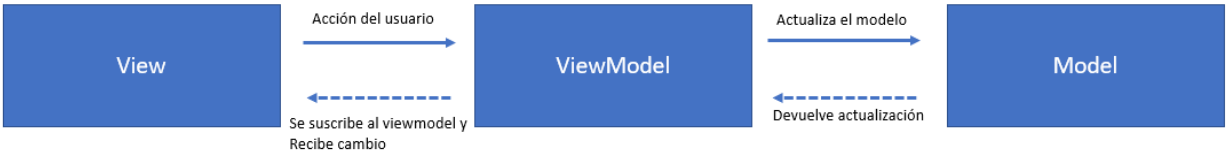


Figura 3 Patrón MVVM.

Model

"Representa la parte de datos, es decir, cuando recuperamos de una base de datos o de un servicio web, toda esa información la almacenaremos en modelos de datos." (Kotlin, 2021) Define la lógica de negocio y los datos que se mostrarán en la interfaz de usuario

View

Es la parte de la UI, los XML, las *Activities* y los *Fragments*, es el nivel que se comunica con el cliente con mayor frecuencia y muestra toda la información No existen lógicas de negocio o interfaz de usuario. Estos funcionaran como siempre, ejecutando acciones, por ejemplo, al pulsar un botón, pero no realizarán las acciones, se suscribirán al *ViewModel* y este les dirá cuándo y cómo pintar. (Kotlin, 2021) Aquí la diferencia con MVP es que el View no tiene lógica, solamente como se comenta, le dirá al *ViewModel* que acción ejecuto el usuario.

ViewModel

Este sería la conexión entre el *Model* y la View. Las View se suscriben a sus respectivos *ViewModel* y estos al percatarse de que el modelo se ha modificado lo notificarán a la interfaz de usuario. (Kotlin, 2021)

Contiene toda la lógica de presentación de la View. Envía comandos al modelo y a notificaciones a la View de los cambios ocurridos en el *Model*. Podemos decir que el *ViewModel* conecta todo el sistema y separa el *Views* y el *Model*. Este tipo de diseño consigue el objetivo de alta cohesión y bajo acoplamiento. La capa de *ViewModel* es la capa de enlace entre el View y *Model*.

El *ViewModel* es simplemente un modelo de la vista. Define e implementa los eventos de entrada del usuario, transforma los datos del *Model* y los prepara para que los muestre la View.

Es importante destacar que *ViewModel* debe ignorar por completo su View correspondiente. No se debe declarar ni utilizar ninguna referencia a una View dentro del *ViewModel*. Debe estar listo en cualquier momento para usarse con cualquier otra entidad de View sin romper el código.

MVVM vs MVP

Como se ha mencionado, ambos patrones cuentan con algunas similitudes entre sí, pero al momento de realizar la implementación de estos, cada uno cuenta con sus diferencias que nos llevan a realizar más cosas que hacen ver su gran diferencia a la hora de desarrollar.

Una de las diferencias es que MVVM utiliza DataBinding, por lo tanto, es una arquitectura que mayormente utiliza eventos, por lo tanto, facilita la separación de la vista con la lógica comercial principal.

MVP por lo general tiene una asignación uno a uno entre el *Presenter* y la View, mientras que MVVM puede asignar muchas *Views* a un *ViewModel*.

El patrón MVP es ideal para aplicaciones simples y complejas, mientras que MVVM no es ideal para proyectos de pequeña escala, más bien tiene mayor sentido implementarlo en proyectos grandes así será más fácil de manejar, estructurar y codificar.

Otra ventaja de MVVM es que su capacidad para realizar pruebas unitarias es mayor, debido a que MVP por tener un estrecho vínculo entre el View y el *Presenter* lo puede dificultar.

En comparación con el patrón de diseño MVC convencional, el patrón de diseño MVVM separa claramente las páginas y la lógica de las páginas.

Ventajas MVP

Algunas de las ventajas de implementar MVP son las siguientes:

- Es sencillo depurar las aplicaciones debido a que MVP presenta tres capas diferentes de abstracción. Aquí es posible realizar pruebas unitarias mientras se desarrolla la aplicación, ya que la lógica comercial es completamente independiente de la View.
- Puede tener múltiples *Presenters* para controlar sus *Views* y así el código se puede reutilizar. Esto puede ser bastante beneficioso ya que no tendrá los planes de tener un solo presentador solo para controlar diferentes *Views*
- MVP funciona manteniendo la lógica empresarial y la lógica de persistencia separadas de las clases *Activity* y *Fragment*. Esto asegura que las preocupaciones se separen de una mejor manera.

Ventajas MVVM

- La mantenibilidad es alta, será posible entrar en partes más pequeñas y enfocadas de los códigos y hacer cambios fácilmente debido a la separación de diferentes tipos de código con una manera más limpia. Esto lo ayudará a avanzar rápidamente con los nuevos lanzamientos y mantenerse ágil.
- Hace pruebas es sencillo, todas y cada una de las partes del código son granulares. Todas las dependencias internas y externas permanecerán desde el código que contiene la lógica central de la aplicación que se tenía planeando probar. Escribir pruebas unitarias contra la lógica central, se vuelve mucho más fácil.
- La extensibilidad es fundamental y con MVVM es posible aumentar y agregar piezas de código granulares o reemplazar las existentes sin ningún problema.

Existen grandes ventajas entre uno y otro, como se comentó, entre si son parecidos, pero la pregunta es, ¿cuál patrón se debería implementar para una aplicación móvil?

Esto depende de varias cosas, como qué tipo de aplicación va a ser desarrollada, así la magnitud del proyecto. También hay que tomar en cuenta que MVVM es un patrón poco más complicado de implementar en un principio.

Si se tiene conocimiento previo de MVC y la aplicación será desarrollado en java, será más fácil entender e implementar MVP.

Por otro lado, lo más recomendable es utilizar MVVM si se va a desarrollar sin conocimientos sobre algún lenguaje para aplicaciones Android, ya que por lo general con MVVM se utiliza *Kotlin*, este siendo más sencillo en su aprendizaje como lenguaje de programación, al igual es el patrón que se recomienda utilizar y actualmente es el más utilizado.

5. Arquitectura limpia con MVVM

Para demostrar el uso de la arquitectura limpia el ejemplo de implementación será el inicio de sesión en una aplicación móvil, más que nada porque lo importante aquí es entender bien el cómo implementar y usar la arquitectura limpia en conjunto con MVVM en un proyecto de aplicaciones móviles Android.

Como se mencionaba anteriormente, arquitectura limpia menciona a las capas interiores y exteriores, las capas internas no deben saber nada de las externas. Esto significa que una capa externa puede contar con alguna dependencia de una capa interna, pero por ningún motivo puede ser el caso contrario.

Tomando esto como base, teniendo las capas de arquitectura limpia, las dividiremos en tres capas principales para nuestro proyecto: capa de presentación, datos y dominio. Como se muestra la Figura 4:

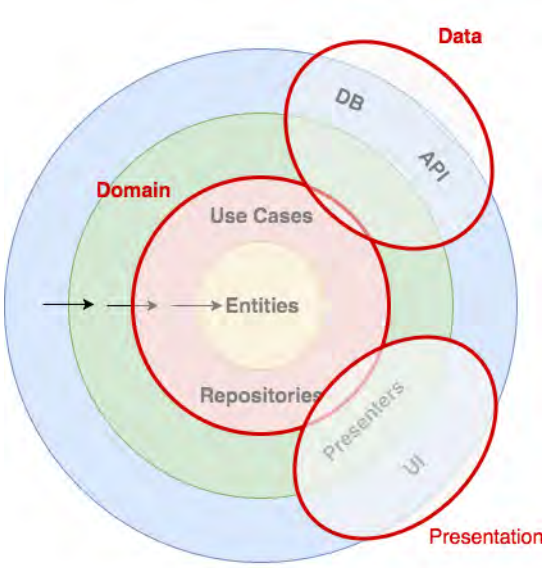


Figura 4 División arquitectura limpia

Teniendo lo anterior en cuenta, la única capa independiente será la de dominio, por otro lado, la capa de datos y presentación, dependerán de lo que se haga en la de

dominio para poder efectuar acciones ya sea para el usuario o los datos empleados en la aplicación.

Ahora, MVVM aquí juega un papel importante para hacer aún más manejable e independiente una capa de la otra en nuestro proyecto, debido a que las *Views* son independientes del *ViewModel* y a su vez el *ViewModel* de los *Models*, solo haría falta ver dónde colocar cada uno de estos en las capas presentadas anteriormente.

Cabe aclarar que el proyecto igual contará con otras capas/carpetas que nos servirán para mejor gestión y entendimiento del proyecto, y evitar más dependencia entre capas.

Después de haber creado un proyecto en Android Studio y tener nuestro proyecto base, lo que haremos es crear 6 principales carpetas para ir almacenando nuestros diferentes archivos.

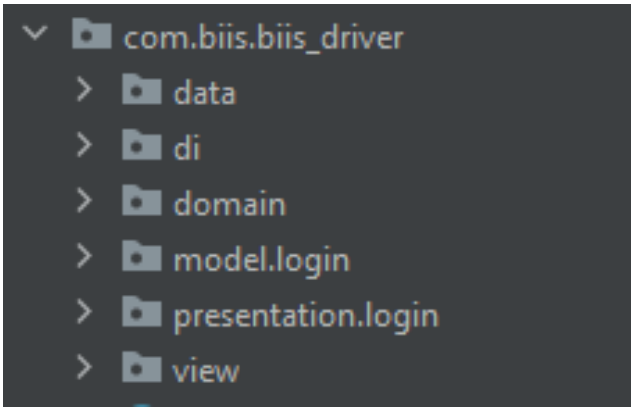


Figura 5 Carpetas del proyecto

De acuerdo con las buenas prácticas del desarrollo, nombramos todas las carpetas en inglés, data, di (*dependencies injection*), *domain*, *model*, *presentation* y *view*.

Como se puede ver, se crearon tres carpetas nombradas como las capas que se mencionaron anteriormente, con esto ya inicia la implementación de la arquitectura limpia. Se obtienen las carpetas *data*, *domain* y *presentation*.

También hay que considerar a la carpeta *di* como parte de la capa de *data*, así como *model* de la capa de *dominio* y *view* parte de la capa de *presentación*.

Capa de datos

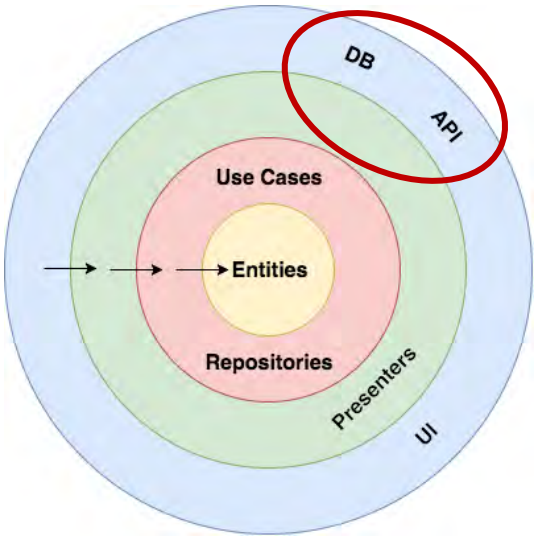


Figura 6 Arquitectura limpia - Capa de datos

En la capa de datos se contendrán las carpetas de *data* y *di*. Esta capa tendrá la función principal de proveer los datos e información necesaria para uso de la aplicación.

Como se puede ver igual en la Figura 6, y de la manera en que dividimos la arquitectura limpia, esta capa es la que tendrá conexión directa a las bases de datos, ya sea locales o bien, consultas a APIs para traer información y datos desde

un *backend*. También existen en Android otras formas de guardar información de manera local, existe el *localstorage*, donde se almacena cualquier dato en formato *json* y se nombra con un identificador.

Para el caso de la consulta hacia *endpoint* de APIs, *Android* tiene un complemento llamado *Retrofit*, que facilita las consultas a cualquier *endpoint*, solo es necesario instalar el complemento e implementarlo.

También contendrá la parte de inyección de dependencias, esto permitirá que cada capa se conecte, y así, por ejemplo, una clase de la capa de datos pueda ser llamada y usada desde la capa de dominio. Para esto se utiliza otro complemento de Android llamado *Dagger*, que sirve para la inyección de dependencias y usar funciones de una clase en otra.

Y esto es importante para la separación en capas de todo el proyecto, de no ser así todo lo tendríamos en un mismo archivo, la lógica de negocio, consultas a *endpoints* guardado de datos, entre otras características.

En el caso del proyecto o nuestra aplicación de inicio de sesión, la responsabilidad de esta capa es enviar la información ingresada por el usuario, como lo sería su nombre de usuario y contraseña, hacia algún *endpoint* para poder validar si los datos ingresados son correctos y así, determinar si el usuario podrá iniciar sesión o no.

En términos generales acerca del patrón MVVM, aquí sería la parte donde se alojaría el *Model*, ya que en esta capa es donde se “manejan” todos los datos necesarios para el funcionamiento de la aplicación.

Podemos notar que tal cual no existe alguna carpeta o algo que haga alusión o directamente se llame *Model*, pero como bien sabemos, en MVVM, el *Model* se encarga de manejar todos los datos e información, y en esta capa, es justo donde tenemos todo esto.

Capa de dominio

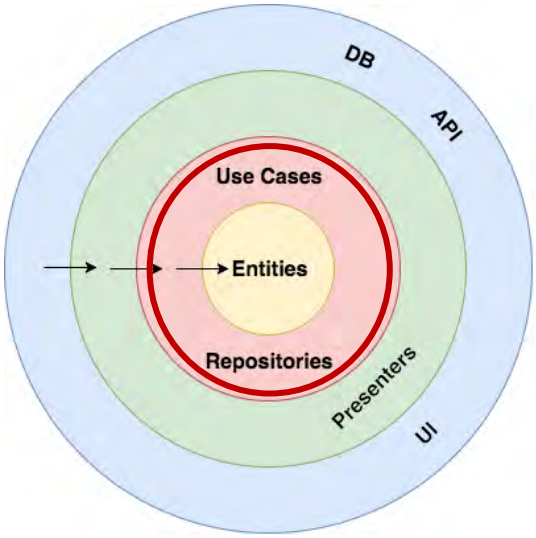


Figura 7 Arquitectura limpia - Capa de dominio

En esta capa, adjuntaremos dos propiedades de la arquitectura limpia, las entidades y los casos de uso.

Básicamente, contendrá la lógica que se desarrollará para las reglas de negocio y como lo dice su nombre, casos de uso que se aplicarán, con el ejemplo de nuestro proyecto, aquí estarían todas las validaciones necesarias para poder permitir o denegar al usuario el inicio de sesión.

Un ejemplo de esto sería si la respuesta que nos dé el *endpoint* consultado en la capa de datos, la usaremos para determinar lo anterior, teniendo en cuenta que el *endpoint* también se encarga de las validaciones en la base de datos, para saber si la información del usuario es correcta. Ya con la respuesta del *endpoint*, la capa de demonio la usará para informarle a la siguiente capa (Capa de presentación) que debe mostrarle al usuario, ya sea una respuesta satisfactoria o no, que sería denegar el inicio de sesión.

Esta capa igual es parte del *Model*, ya que como se mencionó anteriormente, el *Model* igual es responsable de realizar validaciones, manejo de los datos, lógica y reglas de negocio.

Capa de presentación

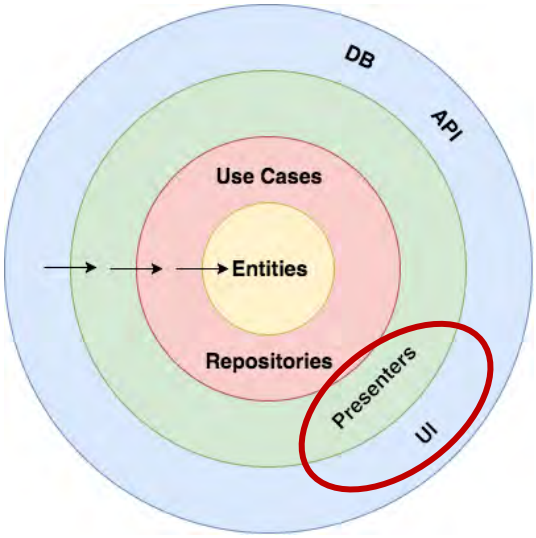


Figura 8 Arquitectura limpia - Capa de presentación

En esta capa se presentará la información al usuario y permitirá actualizar la interfaz de usuario.

Teniendo en cuenta la arquitectura limpia, esta capa exterior, depende de la interior (Capa de dominio) para poder actualizar información en pantalla, determina qué mostrar dependiendo lo que el usuario haga, etc.

Como se puede ver en la imagen aquí se alojará el *presenter* de arquitectura limpia, que en nuestro caso usando MVVM, será el *ViewModel*. Esta se encargará sólo de la conexión entre la View y el *Model*, pasando información o diciéndole que hacer al *Model*, dependiendo de lo que el usuario realice en la *View*, siendo la *View* parte de esta misma capa, y el *Model* de la anterior.

La *View* es representada por *Activities* o *Fragments*, solo se encargará de mostrar u ocultar al usuario cosas de las interfaces de usuario (que son generados con XML). Este podrá contener lógica, pero solamente para hacer cosas en la interfaz, y no se ocupará de la lógica de negocio. Le informará al *ViewModel* por medio de

Data *Binding*, que es lo que el usuario realizó y así el *ViewModel*, solicitará a la capa *Model* realizar toda la lógica de negocio necesaria.

Con el ejemplo del inicio de sesión, lo que hará la View es obtener los datos que ingrese el usuario, su nombre de usuario y contraseña, y cuando este presione el botón de “Iniciar Sesión”, le informará al *ViewModel* que realizó esta acción el usuario, le pasará los datos ingresados, y el *ViewModel* a su vez, pasará estos datos al *Model*, para que haga los necesario para determinar si el usuario podrá iniciar sesión o no.

6. Implementación de Arquitectura limpia con MVVM

Para construir la aplicación se utilizó la estructura de carpetas que se mencionaron en los capítulos anteriores y se podrán agregar nuevas carpetas dentro de la estructura de ser necesario, para darle aún más modularidad e independencia a cada capa.

Implementación de la capa de presentación

En esta capa tendremos las dos carpetas principales, *presentation* y *view*. Aquí estará lo necesario para poder mostrarle al usuario en pantalla lo que necesite, ya sea información, obtener datos del mismo usuario, hacerlo cambiar de una pantalla a otra, etc.

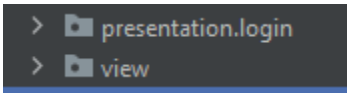


Figura 9 Carpetas capa presentación

Carpeta view

Primero tendremos la carpeta *view*, que contendrá toda la parte del *View* de MVVM. En esta carpeta se sitúan todas las *Activities* (como se mencionó anteriormente igual pueden tener *Fragments*).

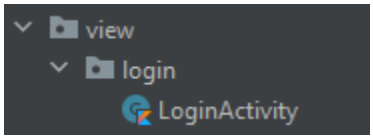


Figura 10 Carpeta view

Tendremos una subcarpeta, lo que hacemos es dividir por pantallas las carpetas. En este caso tenemos la carpeta *login* con el *LoginActivity*, donde tendremos toda la lógica sobre la interfaz de usuario, que es el XML.

Por ejemplo, la función que se muestra en la Figura 11, que lo que hace es obtener la versión de la aplicación desde la configuración de la aplicación y una vez obtenido el dato, mostrar en la pantalla la versión actual de la aplicación.

```
/**
 * Set the version app
 */
private fun setVersionApp() {
    val versionName = BuildConfig.VERSION_NAME
    val version = "Versión" + " " + versionName
    binding.version.text = version
}
```

Figura 11 Función para mostrar versión aplicación

En la Figura 11 se observa cómo la función aplica la arquitectura limpia, ya que su ejecución depende de lo que suceda en la capa de dominio, esperará una respuesta positiva o negativa de esta capa. En caso de ser positiva, envía al usuario a la pantalla de inicio, por el contrario, y ser una respuesta negativa, mostrará al usuario una alerta de error.

```
/**
 * Observe when successLogin changes and validate if it is necessary to change the page
 */
private fun onSuccessLogin() {
    viewModel.successLogin.observe( owner: this, Observer { successLogin ->
        if (successLogin) {
            val intent = Intent( packageContext: this, TripsActivity::class.java)
            startActivity(intent)
        } else {
            alertService.showGenericAlertDialog(
                    LoginConstants.LOGIN_ALERT_TITLE,
                    LoginConstants.LOGIN_ALERT_MESSAGE
            )
        }
    })
}
```

Figura 12 Función ejecutada después de iniciar sesión

Carpeta presentation

En la carpeta *presentation* se situarán todos los *ViewModel*. Así como en la carpeta *view*, tendremos una subcarpeta *login*, donde pondremos el *LoginViewModel*.

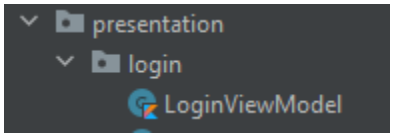


Figura 13 Carpeta presentation

Lo que hará el *LoginViewModel* es estar esperando a que el usuario haga algo para actuar.

```
/**
 * Call loginUseCase to do the login process and assign the response to successLogin
 */
fun doLogin() {
    CoroutineScope(Dispatchers.IO).launch {  this: CoroutineScope
        try {
            _status.postValue( value: true)
            val payload = LoginInterface()
            payload.username = username.value
            payload.password = password.value
            val response = loginUseCase?.doLoginProcess(payload, region.value.toString())
            _status.postValue( value: false)
            _successLogin.postValue(response)
        } catch (e: UnknownError) {
            _status.postValue( value: false)
        }
    }
}
```

Figura 14 Función LoginViewModel

En el caso de la función *doLogin* mostrada en la Figura 14, lo que estará esperando es ser ejecutada mediante *DataBinding* desde el el XML. Esto se ejecutará cuando el usuario le dé clic al botón de “Iniciar Sesión”.

De igual manera con *DataBinding* obtendremos el nombre de usuario (*username* en la función) y su contraseña (*password* en la función).

Estos dos datos pasarán hacia la capa de datos, donde se encuentra las reglas de negocio y consultas a las APIs, para poder validar si los datos ingresados son correctos y así, devolverle una respuesta a la View (*LoginActivity*), y se envíe la respuesta que corresponda al usuario.

Como se puede notar en el patrón MVVM, lo único que hace el *LoginViewModel*, es actuar como un intermediario entre la View (*LoginActivity*) y el *Model* (Capa de datos), para transferir los datos e interacciones del usuario con la aplicación.

Implementación de la capa de dominio

En estas dos carpetas, la de *domain* y *model*, se situará toda la lógica de negocio y aplicación, principalmente en la carpeta *domain*.

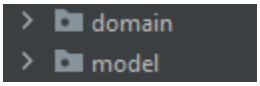


Figura 15 Carpetas capa de dominio

Carpeta model

Esta carpeta tiene el nombre de una parte del MVVM, pero para uso del proyecto, no tiene nada que ver con *Model* de MVVM.

Aquí se situarán los modelos de datos necesarios para poder manipular la información proveniente de las APIs que sean consultadas.

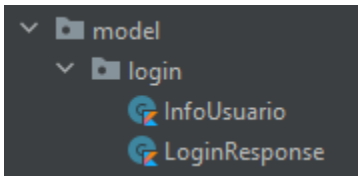


Figura 16 Carpeta model

Como se puede ver contamos con dos modelos de datos: *LoginResponse*, que como dice su nombre, es el modelo de los datos que recibimos como respuesta a la consulta que se hace a un *endpoint* para validar si el usuario y contraseña ingresados son correctos.

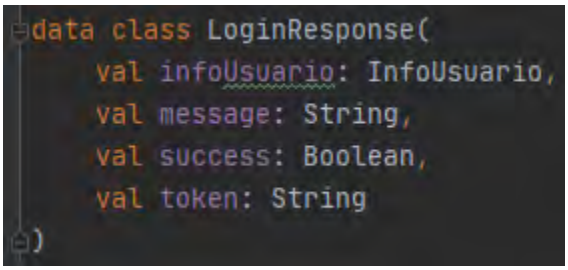


Figura 17 Modelo de datos LoginResponse

Como se puede ver en la Figura 17, como respuesta de la consulta al *endpoint*, esperamos cuatro datos, este modelo nos ayudará a manipular fácilmente estos datos. De igual manera se puede notar que el dato *infoUsuario* tiene como tipo de dato el otro modelo de datos mostrado en la Figura 16, *InfoUsuario*.

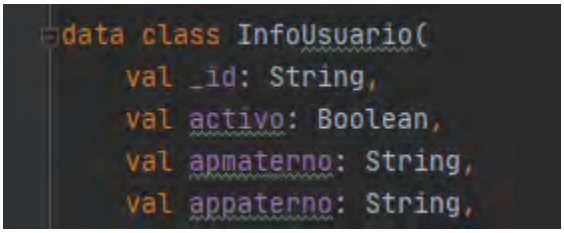


Figura 18 Modelo de datos InfoUsuario

Este modelo de datos contendrá toda la información del usuario, como por ejemplo un identificador único, sus apellidos, entre otros datos.

Estos modelos se utilizarán para manipular fácilmente lo que devuelva el *endpoint*, ya que este responde en un formato JSON, pero con los modelos, los podremos utilizar de manera sencilla donde los necesitemos.

Carpeta domain

Aquí tendremos todo sobre las reglas de negocio necesarias para la aplicación, funciones para guardar datos, validar los mismos, hacer cálculos, etc.

Figura 19 Carpeta domain

Aquí se colocarán dos archivos, el más importante *LoginUseCase*, en este se encuentran todas las funciones para las reglas y lógica de negocio, en *LoginConstants* básicamente se alojarán constantes para mantener un código limpio.

```
/**
 * Call loginRepository to do the login process and return validations
 * @param payload The object to send io the endpoint
 * @return results of validation as true or false
 */
suspend fun doLoginProcess(payload: LoginInterface, region: String): Boolean? {
    saveRegion(region)
    val (loginResp, isSuccess) = loginRepository.doLoginProcess(payload)
    if (!isSuccess) {
        return false
    }
    return validateLogin(loginResp!!)
}
```

Figura 20 Función doLoginProcess

La función de la Figura 20 se encuentra en *LoginUseCase*, esta función se encarga del proceso para validar el inicio de sesión del usuario.

Aquí se utiliza una función de la capa de datos (*loginRepositoy.doLoginProcess*), que es donde se hace la consulta al *endpoint* para validar los datos que el usuario ingreso y fueron proporcionados desde el *LoginViewModel*.

Al obtener una respuesta de esta función se hace una validación con el *IsSuccess*, que nos dice si fue válida la información ingresada por el usuario, de ser negativo retorna un false al *LoginViewModel* y en caso contrario ejecuta otra función del mismo archivo que hará o aplicará otra regla de negocio.

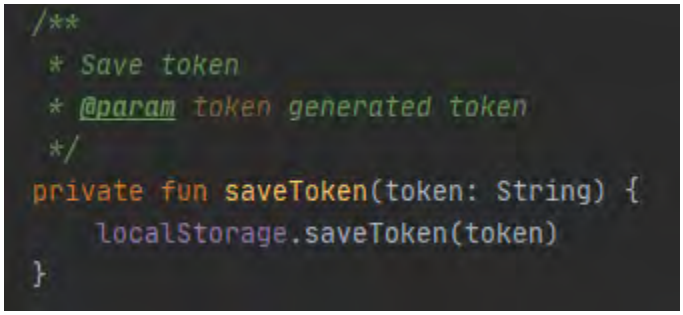
```
/**
 * Validations to verify if is correct user
 * @param loginResp The object with the response of login endpoint
 * @return results of validation as true or false
 */
private fun validateLogin(loginResp: LoginResponse): Boolean {
    val userInfo = loginResp.infoUsuario

    if (userInfo.tipoUsuario.nombre == UserConstants.SHIPPER ||
        userInfo.tipoUsuario.nombre == UserConstants.CARRIER) {
        saveToken(loginResp.token)
        saveUserInfo(userInfo)
        return true
    }
    return false
}
```

Figura 21 Función validateLogin

La función *validateLogin* mostrada en la Figura 21 usará la respuesta del *endpoint* para hacer una validación con la misma. Como se observa, utilizamos el modelo *LoginResponse* para manipular de manera más sencilla los datos que contiene.

En el sistema existen varios tipos de usuarios, pero en la aplicación móvil solo pueden acceder dos tipos de usuarios, entonces con los datos devueltos por el *endpoint*, podemos obtener el tipo de usuario que está intentando iniciar sesión. Si el tipo de usuario es válido retornará un true al *LoginViewModel*, si no es el tipo de usuario válido retornara un false, con esto el *LoginViewModel* le podrá informar al *LoginActivity* si el inicio de sesión es válido y dejará al usuario pasar a la pantalla de inicio, y si no le informará que debe mostrarle un mensaje de error al usuario porque los datos que ingresó no fueron correctos.



```
/**
 * Save token
 * @param token generated token
 */
private fun saveToken(token: String) {
    localStorage.saveToken(token)
}
```

Figura 22 Función saveToken

Después de tener un inicio de sesión correcto, se ejecuta la función *saveToken*, que básicamente lo que hace es guardar el token generado para el usuario de forma local. El token no lo guarda directamente en esta clase, sino que utiliza una función de la clase *localStorage* para guardar la información. Esto se debe a la inyección de dependencias (que será abordada más a fondo en la sección de la capa de datos), y aquí seguimos separando con la arquitectura limpia la funcionalidad de la aplicación, ya que la clase *localStorage* se encuentra en la capa de datos.

Implementación de la capa de datos

Por último, tenemos la capa de datos, en esta capa se encuentran las carpetas de *data* y *di*. En la capa de datos tendremos todo sobre la información y datos necesarios para la aplicación. En esta capa se almacenará la inyección de dependencias y otras funcionalidades nativas para la aplicación como las alertas y *toasters* que se pueden mostrar en la aplicación siendo parte de las dependencias que se pueden inyectar y utilizar.

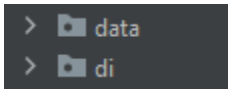


Figura 23 Carpetas capa de datos

Carpeta DI (Inyección de dependencias)

En esta carpeta tendremos toda la implementación sobre la inyección de dependencias. Esto para poder utilizar funciones de una clase en otra como. Por ejemplo, como se mostró en los casos de uso al utilizar desde la clase *LoginUseCase* una función de otra case de esta capa para hacer la consulta al *endpoint*.

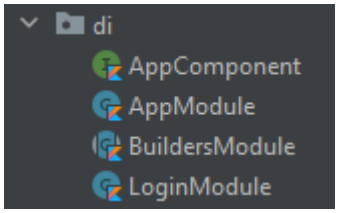


Figura 24 Carpeta di (inyección de dependencias)

Las clases *AppComponent* y *BuildersModule* son parte del mencionado complemento de Android *Dagger*, todo esto siguiendo la documentación proporcionada por el mismo Android para poder configurarlo e implementarlo en el proyecto.

En *AppModule* tendremos la implementación de *dagger* para poder utilizar sobre todo funciones que no son propias de un módulo de la aplicación (como el *Login*, o en otros casos el Inicio, Registro, etc.). Aquí se implementará para poder utilizar las funciones que hacer mostrar alertas y *toasters* o bien las funciones para guardar datos de manera local con una base de datos local o, por ejemplo, con lo mostrado en *LoginUseCase* en la Figura 22, el guardar el Token en *localstorage*.

Por otro lado, en la clase *LoginModule* se implementa de igual forma *Dagger*, pero para hacer uso de este y poder utilizar las funciones de otras clases del módulo de inicio de sesión. Como el ejemplo antes mencionado para consultar *endpoints* para el inicio de sesión.

Carpeta data

En esta carpeta se encuentran todos los servicios que utiliza la aplicación, como se puede ver en la Figura 25, tenemos el servicio de alertas, para poder utilizarlas con la inyección de dependencias en cualquier parte de la aplicación, el *localStorage*, para poder guardar datos en local desde cualquier clase y el *login* donde estará todo sobre el uso de datos del módulo *login* como consulta a *endpoints* o guardar datos en bases de datos locales.

Figura 25 Carpeta data

Por último, tenemos la clase *RetrofitService*, este se encuentra en la raíz de la carpeta data, ya que solo es la implementación del complemento *Retrofit* para hacer las consultas.

También esto es parte del *Model* de MVVM ya que aquí es donde se manejan todos los datos e información de la aplicación y empresa.

Carpeta alertService

En esta carpeta se implementa la funcionalidad en la clase *AlertService* como se puede ver en la Figura 26 para mostrar alertas en pantalla al usuario.

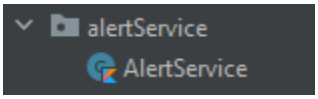


Figura 26 Carpeta alertService

De este modo se evita la declaración o implementación en cada clase de los casos de uso y se invocan las alertas las veces que sean necesarias. Se implementa una vez en esta clase y, por medio de la inyección de dependencias, cualquier otra clase podrá hacer uso de está y mostrar alertas al usuario.

```
fun showGenericAlertDialog(title: String, message: String) {
    val builder = AlertDialog.Builder(activity)
    builder.setTitle(title)
    builder.setMessage(message)
    builder.setPositiveButton(
        GlobalConstants.ACCEPT_ALERT_BUTTON,
        DialogInterface.OnClickListener { dialog, id ->
            dismissDialog()
    })
    alert = builder.create()
    alert?.show()
}
```

Figura 27 Función para mostrar alertas

En la Figura 27 se puede ver la implementación para mostrar alertas en la aplicación, es necesario que al utilizar la función *showGenericAlertDialog*, se envíe el título de la alerta y el mensaje que se quiere mostrar.

```
/**
 * Observe when successLogin changes and validate if it is necessary to change the page
 */
private fun onSuccessLogin() {
    viewModel.successLogin.observe( owner: this, Observer { successLogin ->
        if (successLogin) {
            val intent = Intent( packageContext: this, TripsActivity::class.java)
            startActivity(intent)
        } else {
            alertService.showGenericAlertDialog(
                    LoginConstants.LOGIN_ALERT_TITLE,
                    LoginConstants.LOGIN_ALERT_MESSAGE
            )
        }
    })
}
```

Figura 28 Función de LoginActivity al hacer login

Esta función en el proyecto es utilizada en el *LoginActivity* como se puede ver en la Figura 28, ya que es parte de View en MVVM, aquí en la capa de datos solamente lo estamos implementando, pero se utiliza ahí.

Como se puede ver en la función *onSuccessLogin*, si el inicio de sesión es válido, redirige al usuario a la pantalla de inicio, si no, utiliza la función *showGenericAlertDialog*, utilizando la inyección de dependencias al ser una función de la clase *alertService*.

Carpeta localStorage

En esta carpeta tendremos la implementación de funciones en la clase *LocalStorage*, para almacenar datos. De igual manera que con el *alertService*, lo tenemos aquí como implementación de un servicio para poder ser utilizado en cualquier otra parte de del proyecto.

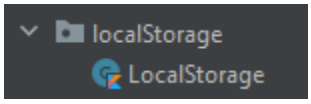


Figura 29 Carpeta localStorage

Aquí tendremos funciones para guardar datos específicos. También se pueden recuperar estos datos guardados.

```
fun saveData(data: String, identifier: String) {
    sharedPreferences.edit().putString(identifier, data).apply()
}

fun getData(identifier: String): String {
    return sharedPreferences.getString(identifier,  defValue: null) ?: ""
}
```

Figura 30 Funciones de localStorage

Con las funciones mostradas en la Figura 30 podremos hacer esto. En la función *saveDate* se guarda el dato. Se envía el dato a almacenar y un identificador con el que podremos recuperar la información. Y con la función *getData*, se enviará el identificador y obtendremos como respuesta el dato almacenado.

Carpeta login

En la carpeta *login* tendremos todo lo necesario para manipular información, ya sea, guardar, obtener, actualizar o editar desde una base de datos local o por medio de *endpoint*. En la carpeta local, se almacenará toda la implementación necesaria para utilizar una base de datos local. En la carpeta *network* estará todo para hacer consultas a *endpoint*.

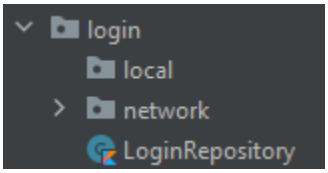


Figura 31 Carpeta login

La clase *LoginRepository* actuará como un intermediario entre los casos de uso (*LoginUseCase*) y cualquier método para manipular información, ya sea con la carpeta local y *network*.

En la Figura 32 podemos ver la función que ejecuta *LoginUseCase* para hacer la consulta al *endpoint* de inicio de sesión.

```kotlin
/**
 * Call loginService to do call login endpoint
 * @param payload The object to send to the endpoint
 * @return loginResponse and if the response was successful
 */
suspend fun doLoginProcess(payload: LoginInterface): Pair<LoginResponse?, Boolean> {
    val loginResp: Response<LoginResponse> = loginService.doLogin(payload)
    val body = loginResp.body()
    if (!loginResp.isSuccessful) { return Pair(null, false) }
    if (!body!!.success) { return Pair(null, false) }
    return Pair(body, true)
}
```

Figura 32 Función doLoginProcess de LoginRepository

Se observa que la inyección de dependencias utiliza la función *doLogin* de la clase *loginService* (que veremos a continuación) y si se puede hacer la consulta devuelve la respuesta del *endpoint* a *LoginUseCase*.

Dentro de la carpeta *network* tenemos toda la implementación para poder hacer una consulta a un *endpoint*. Toda esta implementación siguiendo la documentación del complemento *Retrofit*.

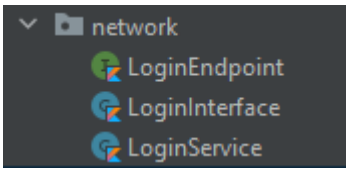


Figura 33 Carpeta network

En la Figura 33 podemos ver tres clases, *LoginEndpoint* es parte de la implementación de *Retrofit*, que hace uso del complemento para hacer la consulta, *LoginInterface* es un tipo de modelo de datos, pero específico de *Retrofit*, para los datos que son necesarios mandar hacia el *endpoint.*

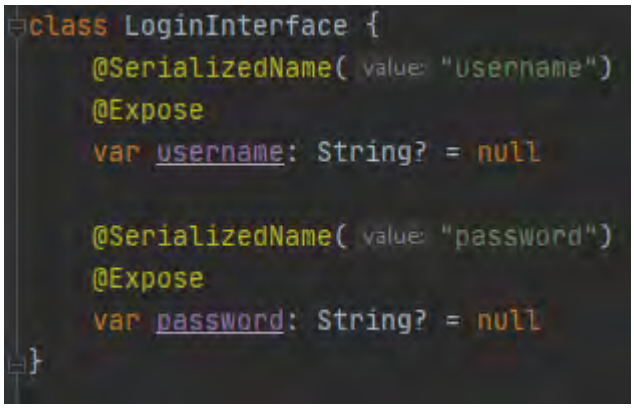


Figura 34 Login Interface

En el caso del inicio de sesión y como se puede ver en la Figura 34, tenemos solamente el username y password del usuario, nombrado así porque es como el endpoint los necesita.

Ahora en la clase *LoginService* tenemos la función *doLogin*, que hace uso *Retrofit* para poder implementarlo en conjunto con la función de la clase *loginEndpoint*.

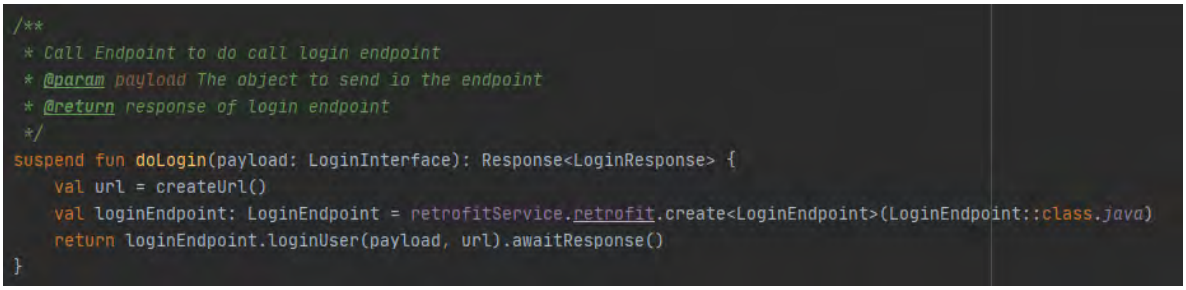


Figura 35 Función doLogin en LoginService

Esta función espera a la respuesta del *endpoint* y regresa al *LoginRepository* para hacer las validaciones vistas anteriormente. Aquí sólo se lleva a cabo la consulta al *endpoint*. Siendo este el último archivo y carpeta del proyecto, ya tendríamos nuestra aplicación de inicio de sesión funcionando, implementada con la arquitectura limpia y utilizando *MVVM*.

7. Conclusiones

Después de investigar sobre la arquitectura para el desarrollo de software, pudimos profundizar más sobre el uso de patrones de desarrollo en Android para incluirlos en proyectos y aplicaciones.

Dependiendo del tipo de aplicación que se desarrollará podremos elegir el patrón que se adapta mejor a un proyecto. Actualmente el patrón MVVM es de los más utilizados y el recomendado por Google.

De igual manera, se describió la arquitectura limpia, para poder aplicar esta metodología en el proyecto y fusionarlo o implementarlo en conjunto con el patrón de desarrollo MVVM.

Como se observó en este trabajo la arquitectura limpia tiene una regla principal: ninguna capa interna puede depender de alguna capa exterior, esto para facilitar el manejo del proyecto, hacer más eficiente el desarrollo y que sea escalable.

La arquitectura limpia divide las tareas en capas, acomodar todo lo referente a consulta de datos y consumo de APIs en una capa, las reglas de negocio en otra y todo lo que tiene que ver con interfaz de usuario en otro nivel.

Con la arquitectura limpia cada capa de la aplicación se encuentra organizada y se tiene acceso al dominio, los controladores y la presentación; por lo tanto, al implementar la funcionalidad del código y posteriormente modificar alguna característica es más fácil acceder a la capa correspondiente y efectuar los cambios y correcciones requeridas. Con esto se administrará de forma más eficiente el proyecto evitando código duplicado, dependencias innecesarias de módulos, al igual que reescribir código.

Ahora bien, implementar la arquitectura limpia con MVVM nos da aún más independencia entre capas, y nos ayuda a estructurar el proyecto de forma eficiente. A través de las diversas carpetas que hacen referencia tanto a la arquitectura limpia como a MVVM se ejemplificó un proyecto con una organización modular.

La arquitectura implementada en la monografía puede ser utilizada en cualquier tipo de aplicación, ya que está hecha para eso, desde aplicaciones pequeñas con 3-4 pantallas, hasta proyectos más grandes. Pero donde se observa mayor utilidad de la arquitectura limpia es en aplicaciones o proyectos grandes, ya que al tener una gran cantidad de pantallas que desarrollar, se puede complicar el buscar archivos específicos; pero si se implementa de forma similar a lo explicado en esta monografía, será más fácil y ágil el poder administrar y encontrar todo lo necesario al momento de desarrollar de forma individual o en equipo.

El desarrollador de software tendrá mayor control de todos los archivos y módulos creados en el proyecto; facilita y agiliza el tiempo de desarrollo y es más sencillo localizar archivos, módulos, secciones que sean necesarias modificar, y nuevas funcionalidades que tienen como objetivo mejorar la mantenibilidad del software.

8. Bibliografía

Bass, L., Clements, P., & Kazman, R. (2013). *Software Architecture in Practice.* USA: Addison Wesley Professional.

Brooks, F. (1995). *The Mythical Man-Month.* USA: Addison-Wesley.

Cervantes, H. (2010). *Arquitectura de Software*. Obtenido de SG Buzz: https://sg.com.mx/revista/27/arquitectura-software

Coates, P. (2010). *Programming Architecture.* Abingdon : Routledge.

Garlan, D., & Perry, D. (1995). Guest Editorial on Software Architectures. *IEEE Transactions on Software Engineering*.

Hanmer, R. S. (2013). *Pattern-Oriented Software Architecture For Dummies.* Chichester: John Wiley & Sons, Ltd.

Juarez, I. (2020). *Resumen Fundamentos de Arquitectura de Software*. Obtenido de Platzi: https://platzi.com/tutoriales/1247-arquitectura-software/9248-resumen-fundamentos-de-arquitectura-de-software/

Kotlin, C. (22 de Abril de 2021). *MVVM en Android con Kotlin, LiveData y View Binding – Android Architecture Components*. Obtenido de Curso Kotlin: https://cursokotlin.com/mvvm-en-android-con-kotlin-livedata-y-view-binding-android-architecture-components/

Leiva, A. (2020). *MVP para Android: Cómo organizar la capa de presentación*. Obtenido de Devexperto: https://devexperto.com/mvp-android/

Martin, R. C. (2018). *Clean Architecture, A CRAFTSMAN'S GUIDE TO SOFTWARE STRUCTURE AND DESIGN.* Pearson Education, Inc.

Miguel, R. M. (2014). *Desarrollo de aplicaciones para Android.* Madrid : RA-MA Editoria.

Nudelman, G. (2013). *Android Design Patterns: Interaction Design Solutions for Developers.* Indianapolis: John Wiley & Sons, Inc.

Oussalah, M. C. (2014). *Software Architecture 1.* Great Britain: ISTE Ltd.

Programación y más. (2022). *Android: ¿Qué es MVC, MVP y MVVM?* Obtenido de Programación y más: https://programacionymas.com/blog/android-mvc-mvp-mvvm#:~:text=Model%20View%20Presenter%20(MVP)%20y,enfoque%20es%20superior%20al%20otro.

Sacristán, C. R. (2013). *Programación en Android.* Madrid: Ministerio de Educación y Formación Profesional de España.

Samuel. (9 de Abril de 2019). *MVVM y DataBinding: Patrones de diseño de Android*. Obtenido de Stips: https://stips.wordpress.com/2019/04/09/mvvm-y-databinding-patrones-de-diseno-de-android/

Semegn, A. D. (2011). *Software Architecture and Design for Reliability Predictability,.* Newcastle: Cambridge Scholars Publishing.

Shaw, M., & Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline.* USA: Prentice Hall.

Statista. (2022). *Share of global smartphone shipments by operating system from 2014 to 2023*. Obtenido de https://www.statista.com/statistics/272307/market-share-forecast-for-smartphone-operating-systems/#:~:text=Smartphones%20running%20the%20Android%20operating,percent%20share%20of%20the%20market.

Szyperski, C. (2002). *Component software: beyond object-oriented programming.* Addison-Wesley.

Vázquez, Á. V., & Valero, J. A. (2019). *Android: del diseño de la arquitectura al despliegue profesional.* México: Alfaomega Grupo Editor, S.A. de C.V.

Vespa, L. (23 de Mayo de 2019). *Patrones Arquitectónicos en Android*. Obtenido de Medium: https://medium.com/@vespasoft/patrones-arquitect%C3%B3nicos-en-android-ded39f7a2c10